CHAPTER
SIXTEEN
SYNCHRONIZING RESOURCES AND CELL

This chapter describes Synchronizing Resources (SR) and Cell. These two systems use mechanisms similar to remote procedure calls of Distributed Processes and Ada. They extend these concepts by providing new mechanisms for scheduling communications.

Two of the important dimensions of interprocess communication are the size of the communication channel and the synchronization required of communicators. To communicating processes, the interprocess communication channels appear to be either of bounded size (*buffered communication*) or of unbounded size (*unbuffered communication*). Communication either requires the simultaneous attention of all communicators (*synchronous communication*) or allows the sending and receiving of messages to be temporally disjoint (*asynchronous, send-and-forget messages*). All explicit process systems choose some point in this two-by-two space. Buffered, asynchronous communication is shared storage—for example, Shared Variables. In Shared Variables, the size of the variable is the size of the communication channel. Processes freely read and write the variable, independent of the status of other processes. Message-based systems use unbuffered, asynchronous communication. Examples of such systems include PLITS and Actors. Procedure-call systems, such as Distributed Processes and CSP, use buffered, synchronous communication. In synchronous communication, both processes must attend to the communication; a process never sends a second message until after the first has been received.*

The language Synchronizing Resources synthesizes these three possibilities. SR supports both synchronous and asynchronous communication. Additionally, the structure of SR programs allows sharing storage between certain processes. In some sense, SR merges a multiprocessing system into a distributed environment.

SR is a language designed for operating systems implementation. One important problem of systems development is scheduling. To simplify scheduling, SR has a priority mechanism built into its communication primitives.

SR is the work of Gregory Andrews at the University of Arizona.† He has implemented SR on UNIX-based PDP-11 systems.

Synchronizing Resources extends the ability of a called process to schedule its interactions by priorities. Cell, a system proposed by Abraham Silberschatz of The University of Texas, combines SR's priority mechanisms with powerful internal queueing structures to provide the programmer with even greater control of the scheduling of process activities.

16-1 SYNCHRONIZING RESOURCES

Synchronizing Resources distinguishes between processes and resources. A *process* in SR corresponds to our familiar notion of process—program and storage, capable of executing concurrently with other processes. A *resource* is a collection of processes that can share storage, their shared storage, and an initialization program for the shared storage. Each process is in exactly one resource; each resource has at least one process.

Processes communicate either through the common storage of a resource or by requests to named entries in other processes. These requests can be either synchronous requests (calls) or asynchronous requests (sends). Entries are declared in the **define** command. This declaration can restrict an entry to receive only synchronous communications (**call**) or asynchronous communications (**send**). We illustrate the structure of an SR program with the skeleton of a program for a

* No one designs a model based on unbuffered, synchronous communication. Unbuffered systems are helpful in that they hold unprocessed messages. Synchronization implies that, from the point of view of the sender, each message dispatch is accompanied by an immediate reception. In an unbuffered, synchronous system there are no unprocessed messages to take advantage of the unbounded size of the communication channel.

† This section is based on the preliminary SR design described in "Synchronizing Resources" [Andrews 81a]. SR has since been modified and extended, principally by the addition of an import/export mechanism, named processes, and minor syntactic improvements. These changes are described in Andrews's 1982 article on the mechanisms, design, and implementation of SR [Andrews 82].

bounded buffer.* Entry names declared in a **define** command are visible outside the resource. In resource stack, the entry pop is called stack.pop.†

```
type item = ... ;                -- declaration of the items stored by the buffer
resource producer_resource;      -- the producer and consumer resources
    process producer;
                ⋮
resource consumer_resource;
    process consumer;
                ⋮
resource buffer;
    define
        insert,                  -- entry name for producers
        remove {call};           -- entry name for consumers. This entry
                                    receives only synchronous calls.

    const bufsize = 20;          -- a buffer of size 20
    var
        first, last : integer;
        queue       : array [0..bufsize − 1] of item;
  -- initialization statements
  process intake;
  -- program for process intake
        ... in (m: item) ...
                    ⋮
  end intake

  process outplace;
  -- program for process outplace
        ... in (var m: item) ...
                    ⋮
  end outplace
end buffer
```

As we mentioned above, requests can be either synchronous or asynchronous. One invokes a synchronous request with the command **call** and an asynchronous

* In those cases where the description of SR [Andrews 81a] omits the details of the declarative structure of the language, we have improvised, using a Pascal-like syntax.

† An entry that is the only entry in the program with a given name can be referenced without mentioning its resource. For example, if there are no other push entries in the program, the entry can simply be called push.

request with **send**. **Call** blocks the calling process until its call is handled. Since the calling and called processes are synchronized, a call request allows the called process to reply to the caller. **Send** transmits the request to the entry and allows the sending process to continue. Each takes the syntactic form of a procedure call. Particular entries may be restricted to receiving only **call** or **send** requests. The example above has the remove entry restricted to calls.

A program accepts requests with an **in** command. This command takes a sequence of guarded clauses (Section 2-2). The guards of the **in** command can reference the parameters of the request and sort calls by priority. An **in** command has the form

```
in <operation_command> ▯ ··· ▯ <operation_command> ni
```

where an <operation_command> is

```
<entry_name> <formal_parameter list>
    and <boolean_expression>
    by  <arithmetic_expression> →
        <statement_list>.
```

The **in** command is a guarded input command. The boolean expression is the guard. The use of a guard (**and** <boolean_expression>) and a priority (**by** <arithmetic_expression>) are optional. This use of **and** as a keyword is something of a pun. This **and** joins the entry name and the guard. The guard is a boolean expression and can contain **and**s of its own, whose meaning is logical conjunction. The <arithmetic_expression> is an integer expression; SR does not support a primitive floating type.

The operation command does not accept a message from a clause with a false guard. If there are several messages on an entry, then these messages are ordered by their priority—their value under the **by** expression. The smallest value has the highest priority. In our other systems with guarded input commands (such as Distributed Processes, CSP, and Ada) the guards refer to the internal state of the receiving process. For example, the input entry of a buffer has a false guard when the buffer is full. In SR, the guard and priority expressions can examine the parameters of the request. For example, an empty taxicab process that wants to respond to the closest (Cartesian-closest) calling customer has the **in** statement

```
in customer (cust_x, cust_y: integer) and cab_free
    by (cab_x − cust_x) * (cab_x − cust_x) +
       (cab_y − cust_y) * (cab_y − cust_y) →
  -- code for responding to customer call
```

where cab_x, cab_y, cust_x and cust_y are the Cartesian coordinates of the cab and the requesting customer, respectively.

SR has four sequential programming constructs: (1) a null statement **skip**; (2) assignment; (3) a guarded alternative statement of the form **if** <guarded_command> □ ··· □ <guarded_command> **fi**; and (4) a guarded repetitive command of the form **do** <guarded_command> □ <guarded_command> □ ... **od**. Guarded commands are syntactically identical to Dijkstra's original guarded commands: <boolean_expression> → <statement_list>. The alternative command executes the statement list of one of its clauses with a true guard. If the guards of an alternative command are all false, the process terminates abnormally. The repetitive command repeatedly executes the statement list of a clause with a true guard until all guards are false. It then terminates. SR does not have procedures or functions—processes, though not recursive, are meant to serve instead of procedures. The program for the bounded buffer resource is as follows:

```
resource buffer;
    define
        insert,
        remove {call};
    const bufsize = 20;
    var
        first  : integer;
        last   : integer;
        queue  : array [0..bufsize − 1] of item;
    first := 0;                         -- initialization statements
    last  := 0;

    process intake;
        do true →
            in insert (m: item)
                and not (((last + 1) mod bufsize) = first) →
                    last        := (last + 1) mod bufsize;
                    queue[last] := m
            ni
        od
    end intake

    process outplace;
        do true →
            in remove (var m: item)     -- Entry call parameters can be either
                                           input parameters (no keyword) or
                                           input-output parameters (keyword
                                           var).
                and last ≠ first →
                    first := (first + 1) mod bufsize
```

```
                        m := queue[first];
                ni
            od
        end outplace
end buffer
```

This program does not treat requests as permutations of producer and consumer calls. Instead, a process is devoted to producers and another to consumers. Both processes run concurrently when there are both empty slots and available messages. These processes communicate through their shared storage. Processes intake and outplace interact in only one place — each tests in its input guard whether the other has gotten too far behind, leaving it with a full (or empty) buffer. We do not synchronize the shared-variable updates because every shared variable is set by only one process (and read by the other). We repeat the theme of single-writer variables in other examples.

Scheduling Requests An SR process has some control over the order in which it handles requests. The guards on input commands can reference not only the internal state of the process, but also the values of the parameters of the message. The priority mechanism can sort messages by "importance." The major limitation of the guard and priority mechanisms is the absence of functions. The guard and priority expressions must therefore be simple and loop-free. An important consequence of this simplicity is that their evaluation must terminate.

In addition to being able to sort requests by priority, a process can access the size of its request queue. The expression ?entryname is the number of requests waiting on entry entryname.

Readers-writers Our next example is a program for the fair control of a readers-writers resource. This program uses both memory sharing within a resource and priority examination in guarded commands to ensure fairness. The readers-writers problem requires a manager that gives access to a resource to two different classes of users—readers and writers. Many readers can simultaneously access the resource, but writers must have exclusive control. Of course, a solution to the readers-writers problem is better if it precludes starving readers or writers. This solution not only allows all readers and writers to progress, but also serves readers and writers fairly—waiting writers keep all newly arrived readers from commencing reading; writers are served in the order that they request the resource.

This fair discipline echoes the linearity of timing and clocks. We ensure a fair queue with a form of clock. The manager resource has two processes, a time_stamp process and a guardian process.* Processes that want the resource call process time_stamp and get a numbered "ticket." One can think of this

* We discuss timestamps and their applications in distributed databases in Section 17-1.

ticket as being stamped with the time of the request. After obtaining a ticket, the potential reader or writer calls process guardian to wait for service. Guardian serves callers (roughly) in the order of their ticket numbers, except that the start of a reading cycle entitles all readers with tickets before the current time to read.

```
resource manager;
type request = (wanttoread, wanttowrite);
define
    stamp {call};          -- entry for ticket stamping
    enter {call};          -- entry for the resource
    exit  {send};          -- When a process is through with the resource, it
                              sends a message on the "exit" entry.
var
    num_waiting_readers, num_waiting_writers: integer;
                           -- the number of readers and writers waiting to
                              access the resource
    clock: integer         -- the guardian's timestamp counter


num_waiting_readers := 0;  -- initialization of common storage
num_waiting_writers := 0;
clock               := 0;

process time_stamp;
    do true →
        in stamp (req: request; var ticket: integer) →
            if
                req = wanttoread →
                    num_waiting_readers := num_waiting_readers + 1
            []
                req = wanttowrite →
                    num_waiting_writers := num_waiting_writers + 1;
            fi
            clock  := clock + 1;
            ticket := clock;
        ni
    od
end process

process guardian
    var
        num_readers      : integer;   -- the number of readers and writers
        num_writers      : integer;      currently using the resource
        num_readers_done : integer;   -- the number of readers and writers
        num_writers_done : integer;      finished with the resource
```

```
last_user             : request;
switch_time           : integer     -- timestamp of the last transition
                                       between readers and writers

num_readers       := 0;
num_writers       := 0;
num_readers_done  := 0;
num_writers_done  := 0;
switch_time       := 0;
last_user         := wanttoread ;              -- an arbitrary choice
do true →
    in enter (req: request; ticket: integer)
        and
            (num_writers = 0) and              -- conditions under which
                                                  a reader can read—no
                                                  writer writing and no
                                                  writer waiting too long
              ((req = wanttoread) and
                 (last_user = wanttowrite) or
                 (ticket ≤ switch_time)))
            or
              ((req = wanttowrite) and         -- conditions under which
                                                  a writer can write—no
                                                  reader reading and no
                                                  reader waiting too long
               (num_readers = 0) and
                 ((num_waiting_readers − num_readers_done = 0) or
                  (last_user = wanttoread)))
        by ticket →
            if
                not (last_user = req) →
                    last_user     := req;
                    switch_time  := clock
              □
                last_user = req → skip
            fi;
            if
                req = wanttoread →
                    num_readers:= num_readers + 1
              □
                req = wanttowrite →
                    num_writers  := num_writers + 1
            fi
  □
```

```
                    exit (req: request) →
                        if
                            req = wanttoread →
                                num_readers       := num_readers − 1;
                                num_done_readers  := num_done_readers + 1;
                          []
                            req = wanttowrite →
                                num_writers       := num_writers − 1;
                                num_done_writers  := num_done_writers + 1;
                        fi
            ni
        od
end guardian
end manager
```

Shared variables num_waiting_readers, num_waiting_writers, and clock do not require mutual exclusion because they are set only by process time_stamp. This process serves as a "secretary" for process guardian, filtering and ordering requests before they reach the "executive's" desk. (Of course, the timestamp counter overflows if the manager is used too long.)

To obtain read access to the resource, a process executes

```
call manager.stamp (wanttoread, my_ticket);
call manager.enter (wanttoread, my_ticket);
  -- access the resource
send manager.exit (wanttoread);
```

Variable my_ticket is of type integer. The protocol for a writer is identical except that it requests wanttowrite.

Initialization, Fairness, and Termination An SR program starts by executing the initialization statement of each resource. Each process begins computing after initialization. SR enforces a weak form of fairness on process scheduling: if a process does not block in a **call** or **in** statement then it eventually gets to execute.

A program terminates when all its constituent processes terminate or are blocked. A process reaching the end of its program terminates *normally*; abnormal termination results when all the guards of an alternative command are false.

Traffic lights A *family* of resources is a set of resources that share the same code. An SR program declares a family of resources by providing a subscripted range declaration for the resource. Similarly, a family of processes can be declared inside a resource. For example, the SR program skeleton

```
type intersection = 1 .. maxlight;
resource light [ intersection ];
    type direction = (north, south, east, west) ;
              ⋮
    process arrival_sensor [direction];
              ⋮
```

declares maxlight resources of type light, each with four processes of type arrival_sensor. Within each resource, myresource is the resource index; within each process, myprocess is the process index.

Our final example SR program is a traffic light control system. The city pictured in Figure 16-1 is installing a distributed system to control the traffic lights at each of the numbered intersections. Each intersection has a traffic light and eight road-sensors. Figure 16-2 shows an intersection. The traffic light has two sets of lights, east-west lights and north-south lights. Each set of lights can be red, yellow, or green. Sensors are embedded in the roadway. They detect and signal the passing of vehicles. The two kinds of sensors, arrival sensors and departure sensors, respectively register cars arriving at and departing from the intersection. For the sake of simplicity, we assume that the roads are all two lanes (one lane in each direction) and that a vehicle reaching an intersection can turn in any direction. The traffic flow is such that a car exiting over a departure sensor is likely (but not certain) to continue to the next controlled intersection in that direction.

When the intersection is busy, we want the traffic lights to cycle in the obvious fashion, letting traffic through first in one direction, then the other. Optimally, when the intersection is not busy the light should go to red-red. This

Figure 16-1 City lights.

Figure 16-2 An intersection.

allows the light to turn green immediately for the next car, without requiring a yellow delay. To improve traffic flow at quiet times we send a message from the departure sensors to the next controller. This message informs the controller that there is probably a car coming. If the controller is in a red-red state, it turns the signal green for that car, allowing it to proceed without stopping. If the controller is busy handling real traffic, it ignores the message.

Our traffic light resource consists of ten processes. It has four arrival_sensors, four departure_sensors, a controller, and a secretary. Arrival_sensors detect the arrival of a car at the intersection. Each approach to the intersection has an arrival_sensor. Departure_sensors register the direction of car departure. Each exit of the intersection has a departure_sensor. Process controller coordinates the traffic information and turns the lights on and off, and process secretary receives messages from other intersections about approaching cars. Sensors recognize passing cars as calls over the a_sense and d_sense entries; secretaries receive messages over the approaching entry. The skeleton of our program is as follows:

```
type intersection = 1 .. maxlight;
resource light [ intersection ];
    type
        direction       = (north, south, east, west);
        light_direction = (ns, ew);
  -- shared storage declarations and initializations
    define
        a_sense, d_sense [direction] {call},
        approaching {send};
    process controller;
                   ⋮
    process secretary;
                   ⋮
    process arrival_sensor [direction];
                   ⋮
    process departure_sensor [direction];
                   ⋮
```

Processes controller, secretary, and arrival_sensors communicate by sharing storage. The sensors write variables ns_arriving and ew_arriving when a car arrives from one of those directions; the secretary writes variables ns_coming and ew_coming when it receives messages about approaching cars. Array goingto stores the intersection to which each departure_sensor leads.

```
type goingrec =
    record
        inter : intersection;
        ldir  : light_direction
    end;

var   -- common storage declarations and initializations
    ns_arriving, ew_arriving : boolean;
    ns_coming, ew_coming     : boolean;
    goingto                  : array [direction] of goingrec;

ns_arriving  := false;
ew_arriving  := false;
ns_coming    := false;
ew_coming    := false;
  -- initialization of the "goingto" array
```

An arrival_sensor reads its next call from its entry and marks the corresponding arriving variable. The entries in this example are not subscripted because each entry is implemented by exactly one process.

```
process arrival_sensor [direction];
    do true →
        in a_sense →
            if
                (myprocess = east) or (myprocess = west) →
                    ew_arriving := true
              □
```

```
                (myprocess = north) or (myprocess = south) →
                    ns_arriving := true
            fi
        ni
    od
end arrival_sensor;
```

Departure sensors accept interrupts from the sensor device and send messages to the neighboring intersections.

```
process departure_sensor [direction];
    do true →
        in d_sense →
            send light[goingto[myprocess].inter].approaching
                (goingto[myprocess].ldir)
        ni
    od
end departure_sensor;
```

When the secretary receives a message from another intersection it marks the appropriate coming variable.

```
process secretary;
    do true →
        in approaching (dir: direction) →
            if
                (dir = east) or (dir = west) →
                    ew_coming := true
              □
                (dir = north) or (dir = south) →
                    ns_coming := true
            fi
        ni
    od
end secretary;
```

The program gives priority to cars that have arrived at a sensor. It allows no more than carmax cars through in a particular direction if cars are waiting in the other direction. If there are no "arrived" cars, the light turns for "coming" cars. The light turns yellow for yellow_time seconds and the system allows car_delay seconds for each car to pass over the arrival sensors. We hypothesize two procedures, delay and signal. Delay(n) suspends a calling process for n seconds; signal(direction,color) turns the lights for the given direction to the given color.

```
process controller;
    const
        carmax      = 20;
        yellow_time = 15;
        car_delay   = 5;
    type
        color = (green, yellow, red);
    var
        ew_green, ns_green : boolean;
        carcount           : integer;

    ew_green := false;
    ns_green := false;
    signal (ew, red);
    signal (ns, red);
    carcount := carmax;

    do
        ew_green →
            if
                (ew_arriving and
                    ((carcount > 0) or not ns_arriving)) or
                (not ew_arriving and not ns_arriving
                    and ew_coming and not ns_coming) →
                        ew_arriving := false;
                        ew_coming   := false;
                        carcount    := carcount − 1;
                        delay (car_delay)
              []
                (ns_arriving and
                    ((carcount ≤ 0) or not ew_arriving)) or
                (not ew_arriving and not ns_arriving
                    and ns_coming and not ew_coming) →
                        signal (ew, yellow);
                        delay (yellow_time);
                        signal (ew, red);
                        ew_green  := false;
                        signal (ns, green);
                        ns_green  := true;
                        carcount  := carmax;
                        ns_coming := false;
                        delay (car_delay)
              []
```

Table 16-1 Green east-west successor states

carcount >0	ew_com.	ns_com.	ew_arr. ∧ns_arr.	ew_arr. ∧¬ns_arr.	¬ew_arr. ∧ns_arr.	¬ew_arr. ∧¬ns_arr.
true	true	true	−c	−c	NS	red-red
true	true	false	−c	−c	NS	−c
true	false	true	−c	−c	NS	NS
true	false	false	−c	−c	NS	red-red
false	true	true	NS	−c	NS	red-red
false	true	false	NS	−c	NS	−c
false	false	true	NS	−c	NS	NS
false	false	false	NS	−c	NS	red-red

Key:		
	−c	Decrement carcount and leave east-west green.
	NS	Turn east-west red and north-south green.
	red-red	Turn both lights red.

```
                not ew_arriving and not ns_arriving and
                    (ew_coming = ns_coming) →
                        signal (ew, yellow);
                        delay (yellow_time);
                        signal (ew, red);
                        ew_green    := false;
                        carcount    := carmax;
                        ew_coming   := false;
                        ns_coming   := false;
            fi
    □
        ns_green →      -- a similar program for ns_green
                ⋮
    □
        not ew_green and not ns_green →
            if
                ew_arriving or (not ns_arriving and ew_coming) →
                    signal (ew,green);
                    ew_green    := true;
                    ew_arriving := false;
                    ew_coming   := false;
                    carcount    := carmax;
                    delay (car_delay)
```

Figure 16-3 The traffic light state machine.

```
                □
                  ns_arriving or (not ew_arriving and ns_coming) →
                                 ⋮
  -- similarly for north-south
                □
                  not ew_arriving and not ns_arriving
                      and not ew_coming and not ns_coming →
                          skip
              fi
     od
end controller
end light
```

The convoluted logic used above is really a selection of a point in a state space. This selection is based on the current settings of the arriving, coming, and carcount variables. Figure 16-3 shows the program's three-state automaton [augmented by the register carcount (c)]. Each arc of this automaton is a disjunction of conjunctions of the arriving and coming variables, and the value of carcount. Table 16-1 summarizes the successor-state relationship for a green light shining in the east-west direction. The boolean expressions in the alternative commands of the program are minimizations on this table. This control structure can also be achieved using table-lookup techniques or by assigning numeric values to the arriving and coming variables and computing relative priorities.

16-2 CELL

Cell is a model-language hybrid; a proposal to extend other languages with a few additional constructs for multiple-process communication and control. Cell's theme is the effective and efficient synchronization of processes. To support synchronization, Cell provides processes that are proper objects and several priority mechanisms for scheduling process requests.

A Cell program is a finite (but not fixed) set of concurrent processes. The processes are called, not surprisingly, cells. A *cell* is a structured data type; the programmer describes the cell class and can create specific instances of that class. Thus, the declaration*

```
type register = cell (initial: in integer);
var reg: integer;
begin
    reg := initial;
    while true do
        select
            accept set (setval: in integer) do
                reg := setval;
            end;
        or
            accept get (getval: out integer) do
                getval := reg;
            end;
        end;
end.
```

declares a class of objects called registers. A register responds to two different kinds of requests, set and get. Set stores a value in the register; get retrieves the

* Following Silberschatz [Silberschatz 80], we use a form of extended Pascal/Ada for our Cell programs.

last value stored. This declaration does not create any registers. A program or a cell that declares

```
var memory: array [1 .. 100] of register;
```

would have (potentially) 100 cells of type register, each named by an element of array memory. This declaration still does not allocate storage. Instead, when the program executes the statement

```
init memory[5](7);
```

a new process (cell) is created, the value of initial in that process is set to 7, that process is set running, and the fifth element of array memory is set to that new process. **Init** is a forking statement; the creating cell continues processing after executing **init**.

Cell processes are objects. Both **init** and (recursive) lexical elaboration can create new cells. Cell names are of a data type called identity; variables can range over this data type. Any syntactic expression that uses the name of a cell can use a variable of this type. The initial parameter of the register is an example of a cell's *permanent parameters.* Cells can have both **in** and **out** permanent parameters. **In** parameters are passed to the cell at its creation. The cell returns **out** parameters when it terminates.

A simple example of using cell names as values and input and output parameters is an accountant cell. A certain operating system charges jobs for printing by the line and plotting by the vector. These charges vary by the time of day and class of user; for any job these charges are the charges in effect when the job starts. When a job is created it is given the name of an accountant cell. The job routes all its printing and plotting requests through this cell. When the job informs the accountant cell that it is done, the accountant cell returns the total charges to the operating system.

```
type accountant = cell
    (print         : in printer_cell;
     print_cost    : in integer;
     plot          : in plot_cell;
     plot_cost     : in integer;
     total_charges : out integer);
var done: boolean;      -- Done is set when the job is through.
begin
     total_charges := 0;
     done          := false;
     repeat
          select
               accept please_print (ln: in line) do
```

```
                print.act (ln);
                total_charges := total_charges + print_cost;
            end;
        or
            accept please_plot (vec: in vector) do
                plot.act (vec);
                total_charges := total_charges + plot_cost;
            end;
        or
            accept finished do
                done := true;
            end;
        end;
    until done;
end.
```

The operating system, having the declarations

```
type
    prt   = cell ... ;     -- printer cell description
    plt   = cell ... ;     -- plotter cell description
    acct = cell ... ;      -- accountant cell description, above
var
    printer                   : prt;
    plotter                   : plt;
    accountant                : acct;
    print_price, plot_price : integer;
    charges                   : integer;
```

could execute the statements

```
init printer ...;
init plotter ...;
print_price := ...;
plot_price  := ...;
init accountant (printer, print_price, plotter, plot_price, charges);
```

to create an accountant cell. It would pass the name of this cell to the job that needs to print and plot. When the accountant cell terminates (after receiving a finish call from its client) it sets charges to the total charges due.

Scheduling The most significant features of Cell are those that order process scheduling. As should be clear from the preceding examples, Cell bases its communication mechanism on Ada's select and accept statements. Processes direct requests at the entries of other cells. Cell extends the scheduling mechanisms of Ada in four ways: (1) accept statements can restrict requests to be from only a specific cell, (2) a queueing mechanism allows cells to delay calls after they have been accepted, (3) the program can specify a partial order on selection from entry and waiting queues, and (4) like SR, requests waiting in delay queues can be ordered by priority. Cell specifies that there is no shared storage between processes and that parameter passing is by value/result. Other than these changes, Cell is a strict generalization of Ada.

The first extension, the **from** clause, makes communication between calling and called cells more symmetric. An accept clause of the form

accept <entryname> (<parameters>) **from** <cellname> **do** ...

accepts calls from only the named cell. This is useful for server-user dialogues—cells that serve several other cells, but whose communication requires a conversation, not just a single request/response exchange. For example, a printer cell would first accept input from any cell and then restrict its input to be from that cell only. This restriction would continue until the end of the printing job. Since the identity of the calling cell is important information to the called cell, the system provides the primitive function **caller**, which (in the scope of an **accept** statement) yields the name of the calling cell.

The second extension introduces the **await** primitive. An **await** statement has the form **await** <boolean-expression>. **Await** statements occur only in the bodies of **accept** statements that are within **select** statements. If a cell executing the body of an **accept** statement reaches an **await** statement, it evaluates the boolean-expression. If the expression is true, it continues processing. If it is false the rendezvous is delayed. The system creates an *activation record* (closure) describing the point of program execution, the request's parameters, and the local variables of the accept clause. It adds this record to a set associated with the **await** statement. (Unlike the accept queue, the **await** set is not ordered.) The cell then executes the program after the **select** statement. The calling process remains blocked. Silberschatz credits the idea for a construct similar to **await** to Kessels [Kessels 77].

What unblocks the calling process? The **select** statement treats requests waiting at **await** statements within its scope as if they were **accept** clauses. If **select** chooses an **await** statement then its processing continues after the **await** statement. Just as **accept** clauses can have boolean guards, the guard of the **await** statement is its original boolean expression. Since this guard can refer to the local parameters of the **accept** statement, the system must potentially evaluate each element of the **await** set in search of one whose local parameters make the guard true. (Silberschatz hypothesizes that in practice, few **await** guards would mention local parameters, and that most of these would be used only for priority scheduling.) If we think of the **select** statement as also accepting clauses from **await** sets, then the program

```
select
    L1: accept queue_1 (<queue_1_parameters>) do
            <statements_1,1>
        M1: await (<boolean_expression_1>);
            <statements_1,2>
or
    L2: accept queue_2 (<queue_2_parameters>) do
            <statements_2,1>
        M2: await (<boolean_expression_2>);
            <statements_2,2>
end;
```

is equivalent, from the point of view of a call blocked in an **await** statement, to the program

```
select
    L1:  accept queue_1 (<queue_1_parameters>) do
            <statements_1,1> ...
or
    L2:  accept queue_2 (<queue_2_parameters>) do
            <statements_2,1> ...
or
    M1: when <boolean_expression_1> –> await M1_accept do
            <statements_1,2>
or
    M2: when <boolean_expression_2> –> await M2_accept do
            <statements_2,2>
end;
```

That is, the **await** statement tries to "continue where it left off."

We use the Exchange Functions model (Chapter 7) to illustrate the **await** statement. Briefly, Exchange Functions supports synchronous and immediate bidirectional communication. Communication is directed over channels. Exchange Functions has three communication operations, X, XM, and XR. A call to X on a channel communicates with any other call to that channel. Calls to XM do not communicate with other calls to XM. Calls to XR communicate only if another communication is waiting. If no other communication is ready for an XR, the input value is returned to the task that calls the XR. The channel's task is to pair possible communicators. We assume that this channel passes values of type item. A cell that does channel pairing is as follows:

```
type channel = cell;     -- no input or output parameters
var
    Xwaiting       : boolean;     -- Is an X call waiting?
    XMwaiting   : boolean;    -- Is an XM call waiting?
    AnswerReady : boolean;    -- Is it time to wake a waiting call?
    ansval      : item;

begin
    AnswerReady := false;
    Xwaiting    := false;
    XMwaiting   := false;

    while true do
        select
            when not AnswerReady →
                accept X (inval: in item; outval: out item) do
                    if Xwaiting or XMwaiting then
                        begin
                            outval      := ansval;
                            ansval      := inval;
                            AnswerReady := true;
                        end;
                    else
                        begin
                            Xwaiting    := true;
                            await (AnswerReady);
                            outval      := ansval;
                            AnswerReady := false;
                            Xwaiting    := false;
                        end;
                end; -- accept X

        or    -- Never match an XM with an XM.
            when not AnswerReady and not XMwaiting →
                accept XM (inval: in item; outval: out item) do
                    if Xwaiting then
                        begin
                            outval      := ansval;
                            ansval      := inval;
                            AnswerReady := true;
                        end;
                    else
                        begin
                            XMwaiting   := true;
                            await (AnswerReady);
                            outval      := ansval;
                            AnswerReady := false;
                        XMwaiting     := false;
                    end;
                end; -- accept XM

        or
            when not AnswerReady →
                accept XR (inval: in item; outval: out item) do
                    if Xwaiting or XMwaiting then
                        begin
                            outval        := ansval;
                            ansval        := inval;
                            AnswerReady  := true;
                        end;
                    else
                        outval := inval
                end; -- accept XR
        end; -- select
end. -- channel
```

Since XR (real-time exchange) is a "real-time" operation, we might want the XR calls to have higher priority than the X and XM calls. Cell provides a mechanism for such priorities: One can label **accept** clauses and **await** statements and specify a partial order on these clauses. When a **select** statement has several such choices, it chooses one of the lowest in the partial order. Syntactically, the **order** statement specifies a partial order on the **accept** and **await** statements. Thus, the program

```
order (L1 < L2; L1 < L3; L2 < L4; L3 < L4);
select
    L1: accept ...;
or
    L2: accept ...
        L3: await ...;
        L4: await ...;
end; -- select
```

specifies that **accept** clause L1 is to be taken in preference to all others, that **await** statement L4 is to be given the lowest priority, and that the system is to choose arbitrarily between **accept** clause L2 and **await** statement L3.

The fourth extension to the conventional select semantics provides that a **by** clause in the **await** statement controls the service order of the elements in the **await** sets. The program

```
var p, q: boolean;
        ⋮
```

```
select
    accept
        this_entry (x, y: in integer ...);
                        ⋮
        await (p and q) by (3*x + 4*y);
                        ⋮
```

orders the elements in the **await** set by the lowest value of 3*x+4*y. This use of numeric priorities parallels SR. In SR, priorities are associated with the **accept** clauses; in Cell they are associated with the **await** set.

Termination and calls Cells are created by the declaration and initiation of cell variables. Hence, Cell supports dynamic process creation. A cell *terminates* when (1) it reaches the end of its program, and (2) all the cells it created (its children) have terminated. Cells cannot force the termination of their children. When a cell with **out** parameters terminates, the values of its **out** parameters are returned to its parent.

It is often useful to treat dependent cells not as concurrent processes but as procedures. A cell that executes

```
call <cell-identifier> (<actual parameters>)
```

both initiates the cell <cell-identifier> and waits for its termination.

Perspective

SR and Cell are both proposals that extend Ada's synchronization mechanisms. These extensions significantly reduce the difficulty of programming many common resource control problems. Nevertheless, each system has several important deficiencies.

SR's approach to distributed computing has three distinguishing features: (1) SR mixes shared storage and request-based communication, (2) SR has both synchronous and asynchronous requests, and (3) SR uses numeric priorities for scheduling. Each of these ideas is a positive contribution to controlling distribution. Our quarrel is with the failure to carry these ideas to their logical conclusion. What is missing from SR is a recursive Gestalt. Processes are to replace procedures, but processes cannot communicate with themselves. And the fixed, two-level hierarchy of resources and processes is a structure that cannot be embedded in other program segments. We present a few examples to illustrate these limitations.

Memory sharing SR processes can share storage with processes within their own resource. No other storage sharing is allowed. This structure mimics a physical system of shared-memory multiprocessors communicating over distributed connections. Figure 16-4 shows one such architecture. However, we can imagine alternative architectures. One simple example of such a system is a ring of processors separated by two-port memories (Figure 16-5). In this architecture, memory sharing cannot be described simply in terms of sharing processes within resources. Each processor shares memory with two other processors, but these processors do not share memory with each other.

SR neglects other opportunities for sharing. For example, processes in a resource can share memory but cannot share entries. Shared entries would eliminate the need for many programmer-created buffers. If there are several printers, then processes could send their printing requests to the printing resource instead of to particular printers. Each printer could get the next request by pulling an item from this entry. Thus, shared entry queues would, without any additional mechanism, turn every resource into a producer-consumer buffer.

Process names SR recognizes that providing both synchronous and asynchronous communication facilitates programming. However, useful asynchronous message-based communication requires that processes have names that can be included in messages. The original SR definition [Andrews 81a] omitted such names. The full implementation of SR [Andrews 82] rectifies this deficiency and treats process names as a distinct data type.

Figure 16-4 The archetypical SR architecture.

Figure 16-5 Two port memory processor ring.

Scheduling SR allows process guards and synchronization expressions to examine incoming messages. However, it requires scheduling constraints to be described solely by priorities (and messages deferred by guards). SR assumes that scheduling algorithms can be expressed by a single number. One can imagine stronger mechanisms that would allow comparison of pending messages. SR uses numeric priorities because they are easy to compute and provide a quick determination of the next message.* Of course, some problems do not require such a priority mechanism and can simply omit priority clauses. Other programs are clearly simplified by its existence. But numeric priorities are not the ultimate in scheduling description. Presumably, some problems can profitably use mechanisms beyond numbers.

Additionally, SR neglects including key communication information in its priority mechanism. SR requests are anonymous. The receiver of a call does not know its originator. And though SR provides both synchronous and asynchronous calling mechanisms, an SR process cannot determine how it was called. These attributes are fertile material for priority mechanisms. A process might want to provide better service to a certain class of users or to respond more quickly to requests when the caller is waiting (synchronous requests). An SR program is

* Numeric priorities allow determination of the next message by a simple linear search of the message queue. This search requires storing only the lowest priority value (and a pointer to the message with that value). Partial-order mechanisms require more search and more structure.

forced to encode these concepts in the message where they are subject to error and abuse.

Cell Cell contributes three key ideas: explicit process objects, delay queues in **accept** statements, and partial ordering of choices. Cell has borrowed the foundation for each of these ideas and cleverly extended the concepts. However, Cell also fails to carry these ideas to their logical conclusions.

More than most languages in the imperative, systems tradition, Cell recognizes the desirability of processes as objects — structures that can be created, destroyed, named, and used. Cell's implementation of this idea is almost complete. The major omission is the dynamic creation of cells outside the recursive process—the equivalent of the **new** function in Pascal. What is novel and interesting is the passing of output parameters on cell termination.

Process initialization parameters (or their equivalent) are a familiar concept. In addition to initialization, terminating cells return parameters to their callers. Thus, the program

```
type this_cell = cell (y: out integer) . . . ;
var
    j  : integer;
    c  : this_cell;
           ⋮
    init c(j)
           ⋮
    j := 5;
    write(j);
           ⋮
```

assigns the value computed by terminating cell c to variable j *sometime after cell* c *terminates*. This happens asynchronously with respect to the creating process. That is, there is no way of telling when it will happen. Thus, one cannot be sure that the write statement will actually print 5. In practice, one can avoid such pitfalls by not assigning to output variables. However, designing such traps into the language is a mistake.

The **await** statement integrates the monitor delay queue with the indeterminacy of the Ada **select** statement. Cell wisely recognizes that program state is useful for synchronization and scheduling. We object to Cell's limitation of reviving awaiting requests only at the point of the **await** statement. We propose that an explicit queue of awaiting callers (like the entry queue) that could be accessed at any point in the program would provide a more flexible scheduling structure. (Of course, the various SR and Cell proposals for queue ordering can be retained.) We contrast the Cell program for an Exchange Functions channel that requires continuation (as given above) with the same program that treats await queues as entries. In this program, we name the await queue and explicitly mention its parameters.

```
type channel = cell;    -- no input or output parameters
var
    barXMaccept : boolean;    -- The last accept was an XM.
    barXRaccept : boolean;    -- The last accept was an XR.
    ansval      : item;

begin
    while true do
    begin
        barXMaccept := false;
        barXRaccept := false;
        select
            accept X (inval: in item; outval: out item) do
                ansval := inval;
                await MatchCall (inval, outval);
                   -- Wait in the MatchCall queue.
            end; -- accept X
        or
            accept XM (inval: in item; outval: out item) do
                barXMaccept := true;
                ansval      := inval;
                await MatchCall (inval,outval);
            end; -- accept XM
        or
            accept XR (inval: in item; outval: out item) do
                barXRaccept := true;
                outval      := inval;
            end; -- accept XR
        end; -- select

        if not barXRaccept then
        begin
            select
                accept X (inval: in item; outval: out item) do
                    outval := ansval;
                    ansval := inval;
                end; -- accept X
            or
                when not barXMaccept →
                    accept XM (inval: in item; outval: out item) do
                        outval := ansval;
                    ansval := inval;
                end; -- accept XM
        or
            accept XR (inval: in item; outval: out item) do
                outval := ansval;
                ansval := inval;
            end; -- accept XR
        end; -- select
        accept MatchCall (inval: in item; outval: out item) do
           -- treating the MatchCall await queue as an entry
                outval := ansval;
            end;
    end; -- not an XR
  end; -- while loop
end; -- end cell definition
```

This program implements a three-step algorithm: Receive a call. If that call is an XR, return its value. Otherwise receive a matching call, respond to it, and then complete the first call.

The partial-order priority of the select statement is a clever idea. Conditional statements originally had one determinate form: **if** ... **then** ... **else** ... **if** ... **then** ... **else** .... The programmer specified an order for the evaluation of the conditions; the program followed that order. Guarded commands, a later invention, took the opposite approach: the programmer does not specify any order. The partial-order priority of the Cell select statement encompasses both of these approaches and all points between. Cell requires that this partial order be fixed at compilation. Programming, compilers, and computer architecture being what they are, this is perhaps an inevitable decision. Certainly a system that allows the priorities to shift based on program experience would be more flexible. Such dynamic rearrangement allows the programmer to easily specify priorities such as "select the entry queue that is the busiest." We also regret that Cell limits partial-order guards to select statements. They are an interesting programming structure for languages in general.

PROBLEMS

16-1 Does SR need both synchronous and asynchronous communication mechanisms? Argue whether synchronous communication can be modeled by asynchronous communication, and conversely, if asynchronous communication can be modeled by synchronous communication.

16-2 The SR readers-writers program is only weakly fair, even with respect to the values presented on the timestamps. Why is it not strongly fair?

16-3 Reprogram the readers-writers problem to allow a few readers that arrive after a waiting writer to access the resource on the current read cycle. What criteria can be used to decide what "a few" is? Be sure that your solution precludes starvation.

16-4 Rewrite the readers-writers program to dynamically adjust the service levels of readers and writers based on the historic access patterns to the resource.

16-5 Rewrite the reader-writers controller to use another secretary process to receive exit messages, instead of routing these messages to the primary guardian process.

16-6 Arrange the sensors and controller of the traffic light problem to permit all communication to be through shared memory. Your program must keep every controller in a different resource.

16-7 The traffic light program specifically tests a large boolean expression to decide what to do. The text hints that an arithmetic expression of the basis variables can be used instead. Rewrite the traffic light program to control the signals by a priority index for each direction. Make sure your program has the same behavior as the original.

16-8 Modify the traffic light program to allow special routing for emergency vehicles. Design a protocol for communication between intersections and a technique for describing the intended path for a fire engine or ambulance. Arrange for green lights all along the emergency path well before the arrival and during the passage of the emergency vehicle.

16-9 Are output parameters in Cell really necessary or can they be imitated by some more conventional mechanism? How could a program obtain the same effect?

16-10 Using the **from** clause, write the Cell controller for a printer that can be initiated by any process but serves that process only until the initiating process's job is completed.

16-11 Write the Cell program for an elevator-algorithm disk scheduler. This scheduler treats the disk head as an elevator, moving it from the edge to the center and back again, always trying to serve calls in its path. Calls to this scheduler specify the track to be read; the scheduler returns the data on that track. The elevator algorithm seeks to minimize disk head movement when there are many simultaneous calls on the disk.

16-12 Write the SR program for the elevator-algorithm disk scheduler.

16-13 Contrast the Cell and SR disk schedulers.

16-14 Cell (like SR and Ada) has an attribute of each entry that is a count of the number of pending invocations of that entry. Cell does not have a corresponding system-defined count of pending calls on **await** statements. Why not?

16-15 Rewrite the SR time-stamped readers-writers program in Cell. Instead of a separate time_stamp process, delay callers in an **await** statement in a single resource.

16-16 Contrast the **await** statement in Cell with queues in Concurrent Pascal.

REFERENCES

[**Andrews 81a**] Andrews, G. R., "Synchronizing Resources," *ACM Trans. Program. Lang. Syst.*, vol. 3, no. 4 (October 1981), pp. 405–430. This paper describes SR. Besides detailing the language and providing a few sample programs, it includes comments on implementing SR and on proof rules for SR.

[**Andrews 81b**] Andrews, G. R., "SR: A Language for Distributed Programming," Technical Report TR81-14, Computer Science Department, University of Arizona, Tucson, Arizona (October 1981). This is the SR manual.

[**Andrews 82**] Andrews, G. R., "The Distributed Programming Language SR—Mechanisms, Design and Implementation," *Softw. Pract. Exper.*, vol. 12, no. 8 (1982), pp. 719–753. Andrews presents both an overview of the SR language and a discussion of the implementation issues involved in creating an SR system.

[**Kessels 77**] Kessels, J.L.W., "An Alternative to Event Queues for Synchronization in Monitors," *CACM*, vol. 20, no. 7 (July 1977), pp. 500–503. Kessels proposes a **wait** statement for monitors. Silberschatz drew his inspiration for the **await** statement in Cell from this paper.

[**Silberschatz 80**] Silberschatz, A., "Cell: A Distributed Computing Modularization Concept," Technical Report 155, Department of Computer Science, The University of Texas, Austin, Texas (September 1980). (To appear March 1984, *IEEE Trans. Softw. Eng.*). This is a concise description of Cell.